A Simple Synchronous Distributed-Memory Algorithm for the HPCC RandomAccess Benchmark

Steven J. Plimpton, Ron Brightwell, Courtenay Vaughan, Keith Underwood
Sandia National Laboratories*
Mike Davis
Cray Inc.

Abstract

The RandomAccess benchmark as defined by the High Performance Computing Challenge (HPCC) tests the speed at which a machine can update the elements of a table spread across global system memory, as measured in billions (giga) of updates per second (GUPS). The parallel implementation provided by HPCC typically performs poorly on distributed-memory machines, due to updates requiring numerous small point-to-point messages between processors. We present an alternative algorithm which treats the collection of P processors as a hypercube, aggregating data so that larger messages are sent, and routing individual datums through dimensions of the hypercube to their destination processor. The algorithm's computation (the GUP count) scales linearly with P while its communication overhead scales as $\log_2(P)$*, thus enabling better performance on large numbers of processors. The new algorithm achieves a GUPS rate of 19.98 on 8192 processors of Sandia's Red Storm machine, compared to 1.02 for the HPCC-provided algorithm on 10350 processors. We also illustrate how GUPS performance varies with the benchmark's specification of its "look-ahead" parameter. As expected, parallel performance degrades for small look-ahead values, and improves dramatically for large values.*

1. Introduction

The High Performance Computing Challenge benchmark suite is designed to test machine performance on a variety of small codes representative of kernel computations in scientific applications. The 7 benchmarks are described in [3, 8] and performance data for a variety of machines are given at the HPCC WWW site http://icl.cs.utk.edu/hpcc/index.html. RandomAccess is a test of a machine's ability to access and update global memory. It defines a table (i.e. a vector) of 64-bit datums that fills roughly half the total memory of the machine and requires the table entries be updated in a random order. An update consists of generating a random location in the table and performing a bit-wise XOR of a 64-bit random quantity with the datum stored at that location. On a parallel machine, processors can perform updates simultaneously, but the locations generated by a single processor will be scattered throughout the global table. On a distributed-memory message-passing machine, this implies the generating processor must communicate with the processor owning a specific table location in order to complete the update. Because the update computation is cheap, the latency cost of sending a small message to the owning processor dominates, and parallel performance is poor. Table 1 lists the giga-updates-per-second (GUPS) performance of a Cray XT3 on the standard RandomAccess code (unoptimized category) provided with the HPCC benchmark suite. These runs were performed on the Red Storm machine at Sandia National Labs which consists of 10368 2.0 GHz AMD Opteron processors configured as a 3d mesh, with a torus in the Z dimension. The custom Cray interconnect and associated system software provide an application-level message-passing performance of 1.1 GB/s bandwidth per direction with an aggregate 2.2 GB/s bidirectional bandwidth and 7 μsec latency (MPI sends and receives).

The single processor performance (0.0147 GUPS) is limited by the memory bandwidth the Opteron processor can sustain for the update operation while accessing table locations spread randomly throughout local memory (half of 2 GB in this case). The parallel performance of the baseline benchmark is poor on all message-passing systems. In fact, Red Storm's 1.02 GUPS on 10350 processors is the fastest rate for any machine listed at the HPCC WWW site (unoptimized category), despite the fact that it represents a parallel efficiency of less than 1%.

These results have motivated ourselves and others to

*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Processors	1	16	64	1024	5000	8192	10350
GUPS	0.0147	0.0105	0.0634	0.195	0.536	0.797	1.018

Table 1. *Performance of the unoptimized RandomAccess benchmark on a Cray XT3 machine in giga-updates-per-second (GUPS) for varying processor counts.*

devise new algorithms that are better tuned for message-passing environments and for specific machines. The benchmark allows for such optimizations, provided they adhere to the benchmark rules. In the optimized category, the current performance leader (as of May 2006) is the IBM BG/L machine which achieved 35.5 GUPS on 128K processors (as compared to an unoptimized category performance of 0.454 GUPS on 2048 processors and 0.0657 GUPS on 64K processors). The second-place machine was a Cray X1E machine with a GUPS rate of 7.69 on 1008 processors. Unlike BG/L and Red Storm, the X1E has native hardware support for remote loads and stores that gives it an inherent advantage for small, remote transactions.

In the next section, we describe an alternative algorithm for the RandomAccess benchmark that is motivated by the style of collective communication popular a decade or more ago for parallel architectures with processors connected in a hypercube topology. As we explain, within the rules of the benchmark, a collective "all2all" operation can be used with appropriate data filtering and aggregration. The all2all routines provided with standard MPI do not allow for data filtering at the lowest level, and typically do not route messages in the hypercube manner we advocate as optimal for this benchmark. Instead, we have implemented an all2all that is similar to the *crystal_router* algorithm described in [4], as a means of routing data through the dimensions of a hypercube.

Performance and additional communication optimizations for the RandomAccess algorithm we propose are highlighted in subsequent sections. Performance with a variable "look-ahead" parameter as defined by the benchmark is also discussed.

2. Algorithm

The serial RandomAccess benchmark is simple to explain. Assuming an N-length table of 64-bit quantities has been allocated and initialized, M updates are performed as follows, in C-like notation:

```
loop 1 to M:
    ran = (ran << 1) ^ (ran < 0 ? 7 : 0)
    index = ran ^ mask
    table[index] ^= ran
```

The first line of the loop generates a randomized quantity (ran) by operating on a primitive polynomial ($x^{63} + x^2 + x + 1$) over GF(2), the Galois Field of order 2. The second line uses low-order bits in ran as the index of a location in the table. The final line updates the table entry via an XOR operation. Note that all operations are fast bitwise logical operations. The length of the table N (a power-of-two), the iteration count M, the initial table values, the initial random number (ran), and the mask definition are all specified by the benchmark. N is chosen so that the table fills roughly half the processor's memory.

For parallel machines, N is chosen so that the table fills half the aggregate memory of all P processors. Each processor stores an N/P length portion of the global table. The stream of M random values can also be partitioned, so that each processor generates M/P table indices. For large P almost all such indices will be for table locations stored on other processors. On distributed-memory message-passing machines, this requires a message to be sent from the processor that generated the index to the processor owning the table location. The message content is the 64-bit ran value which the receiving processor can use to compute a local table index and perform the update.

The parallel algorithm for RandomAccess provided in the HPCC benchmark suite adds asynchronous MPI communication calls to the serial loop described above. Generation of an off-processor index results in an MPI_Isend of the ran quantity to that processor. These quantities are buffered to avoid sending tiny messages. Incoming messages are polled for via MPI_Test/MPI_Recv and processed as they arrive. This algorithm has the advantage of asynchronicity so that processors can do useful work while waiting for data. It can also be implemented on machines that support one-sided communication (i.e. via UPC). However, on traditional message-passing machines, the algorithm requires many small messages and latency costs dominate (see Table 1).

The definition of the benchmark allows each processor to generate and store a small number of indices before the corresponding updates must be performed. This "look-ahead" parameter, Q, is limited to 1024 values. This motivates a synchronous parallel algorithm, executed by each processor:

```
loop 1 to M/P/Q:
    generate Q datums (ran values)
```

```
Q' datums = all2all(Q datums)
perform Q' updates on local table
```

The first and last lines of the loop perform on-processor computations to generate and use Q (or Q') random values for updates, identical to the operations of the serial algorithm. The second line performs a globally synchronous "all2all" communication. Each processor has Q values to send to other processors. Likewise, it will receive Q' values from other processors. On average, Q' = Q, though randomness will induce small variations. [1] MPI provides an MPI_Alltoallv function, which is typically implemented by each processor sending P-1 messages, one to every other processor (if required). However, because Q is small, for large P this will result in many small messages and be inefficient.

Consider alternative methods of performing an all2all operation, as diagrammed in Figure 1. If P processors are viewed as a 1d list, the operation proceeds as described above. In a single stage, each processor sends a message to (potentially) every other processor. The green processor sends data directly to the red processor. If the P processors are viewed logically as a 2d array, the operation can be performed in 2 stages. Each processor first communicates within its column, then within its row. To send a message from the green processor to red, green first sends to the blue processor and blue then sends to red. Note that this affords a large savings in the number of messages each processor sends. If P = 10000, then in 1d each processor sends 10000 messages to complete the all2all, but in 2d for a 100x100 logical array, each processor sends only 200 messages.

Similarly, for a 3d logical grid of processors, the all2all operation can be completed in 3 stages, green to blue to purple to red, with a further reduction in the number of messages. This was the key insight of IBM for their optimized implementation of the RandomAccess benchmark for the BG/L machine (35.5 GUPS) whose processors are interconnected as a 3d torus [5, 6]. Their algorithm is asynchronous (not an all2all), but individual datums are effectively routed from one processor to another in a 3-stage process as diagrammed in Figure 1.

The asymptotic limit of Figure 1 is to view the processors as an n-dimensional hypercube ($P = 2^n$) and perform the all2all in $n = log_2(P)$ stages. Thus any of the all2all algorithms, including the hypercube limit, can be characterized by a dimensionality d, ranging from 1 to $\log_2(P)$. If we assume the dth root of P is an integer, then all2all communication cost for any of the variants is as follows. The total volume, V, of data sent and received by each processor is $dQ(\sqrt[d]{P}-1)/\sqrt[d]{P}$. The number of messages, L, sent (and received) by each processor is $d(\sqrt[d]{P}-1)$. Thus for a traditional 1d all2all, $V = Q(P-1)/P$ and $L = P-1$. By contrast, in the hypercube limit, since $\sqrt[\log_2(P)]{P} = 2$, then $V = \log_2(P)Q/2$ and $L = \log_2(P)$. Thus, the number of messages sent and received is reduced from $P-1$ in the traditional 1d all2all to $\log_2(P)$. This is the chief advantage of the hypercube-style all2all. The added cost is that V, the total volume of data exchanged, increases by a factor of $log_2(P)/2$, since half the data Q must be sent and received at each stage. Additionally, processors must scan the Q datums at each stage to decide which to keep and which to send to a partner processor; this incurs memory copy costs. For this benchmark, because the aggregated message size is still small (Q/2 = 4K bytes), the savings from fewer messages far outweighs these costs, particularly as P increases, as will be shown in the next section.

The hypercube RandomAccess algorithm is simple to implement with standard MPI send/receive calls. Pseudocode executed on each of a power-of-two number of processors is shown in Figure 2.

The $log_2(P)$ loop over hypercube dimensions replaces the all2all line of the previous parallel algorithm. Each processor (me) identifies its partner processor in the dth dimension and sets a mask value to select the bit that represents that dimension in the processor sub-field of the 64-bit datums. It then loops over the Q datums and splits them into 2 lists. Datums that are kept are those that belong to table locations in the same half of the hypercube that the processor belongs to. Datums to be sent belong to processors in the other half. The send-list is then sent to the partner processor and the received list is appended to the keep-list. After looping over all dimensions the final list contains Q' datums for table locations owned by the processor; it can then perform the updates.

For non-power-of-two processors, the algorithm is slightly more complex. We use a recursive approach where a partition initially contains all the processors, and is successively halved until it has a single processor. Similar to the power-of-two algorithm, at each stage, processors in the partition split the list of datums into two sub-lists, a keep-list for datums that correspond to table locations owned by processors in the same half of the partition the processor belongs to, and a send-list for datums belonging to processors in the other half. If the partition has an even number of processors, each processor partners with one in the other half and the send/receive exchange is performed as in the power-of-two pseudo-code. For an odd-size partition, N processors exchange their send-lists with N+1 processors in the following manner. Each of the N processors in the smaller half

[1] Since $Q' > Q$ can occur, this may seem to violate the benchmark rule that a processor cannot accumulate more than Q updates before processing them. However, it does not violate the spirit of the benchmark (personal communication with Bob Lucas, one of the HPCC organizers). Alternatively the algorithm could simply discard $Q' - Q$ of the datums when $Q' > Q$. In our tests, this is less than 1% of the total and is thus allowed by the benchmark which states that up to 1% of the updates can be "missed" (to allow for shared-memory collisions). We choose not to do this, so this algorithm completes 100% of the generated updates.

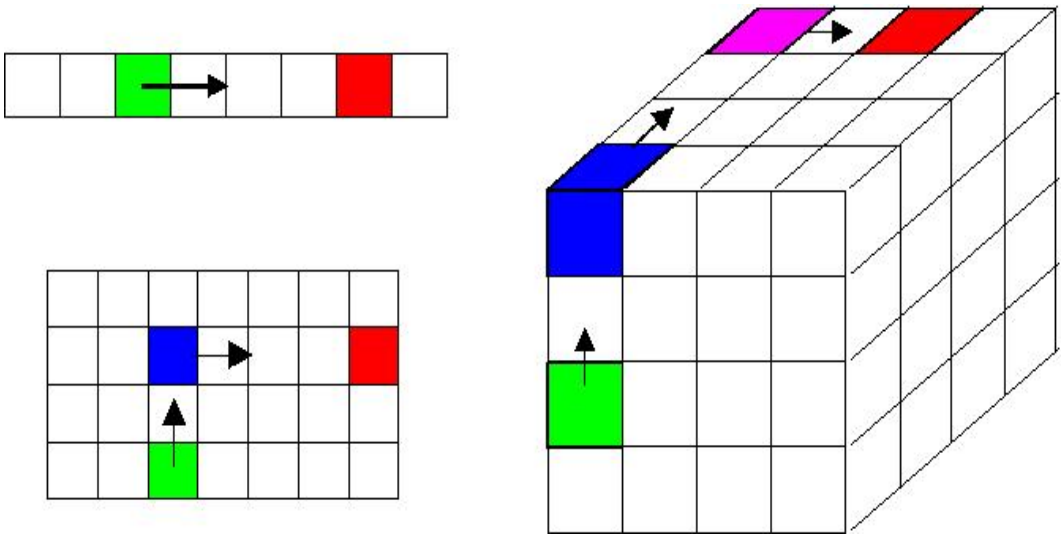

Figure 1. *1d, 2d, and 3d versions of an all2all communication operation. Each square (box) represents a processor. The processors can be logically organized as a 1d list, 2d array, or 3d grid and communicate in stages within each dimension to complete the all2all.*

sends 2 messages and receives 2. Each of the N+1 processors in the larger half also sends and receives 2 messages except for the first and last processor which each send/receive a single message. For load-balancing purposes, the sizes and destinations of these messages are set so that each processor receives the same number of datums (on average) as other processors in its half of the partition.

3. Results

Our implementation of the hypercube RandomAccess algorithm was run on the Red Storm machine with the results shown in Table 2. The first line lists the original (unoptimized) GUPS rates. The second line lists rates for the new algorithm. The one-processor timing shows a slight improvement, because the new code was built with a different compiler, optimization flags, and MPI library.

Note that the hypercube algorithm significantly improves performance — even on small numbers of processors. The non-power-of-two results do not scale quite as well as their power-of-two neighbors, due to the extra messages required in the hypercube-style communication, as described above. For large P, the GUPS rate nearly doubles each time P is doubled. This is in accord with the scaling properties of the hypercube algorithm. When $P_2 = 2 * P_1$ processors are used the GUP count doubles (since it is linear in P), while the additional communication overhead is an extra iteration of the hypercube loop over dimensions. In other words, the communication cost grows logarithmically in P; in this case it increases by only a factor of $\log_2(P_2)/\log_2(P_1)$, which is nearly one for large P_1. Thus with this algorithm the only limitation to a high GUPS rate is the number of processors a machine has, assuming a very large machine still has sufficient bisection bandwidth to perform exchanges of 4K-byte messages between all pairs of processors at each dimension of the hypercube loop.

The third line of Table 2 lists predicted GUPS rates using a model equation that reflects these two scaling factors, namely $T_P = T_1 + T_C \log_2(P)$. T_P is the CPU time to run on P processors and is the sum of two terms. T_1 is the time to run on a single processor which is solely due to on-processor computation (the generation and update steps of the pseudo-code listing). The second term represents the logarithmic cost of the all2all communication. T_C is the time to exchange a 4K-byte message between a pair of processors, and also includes the cost of scanning the Q datums to copy them to send- and keep-lists. On Red Storm we use values of T_1 = 56.9 μsec (for 1024 updates), and T_C = 33 μsec. The latter is consistent with MPI latency/bandwidth costs for 4K-byte messages, measured in independent tests on Red Storm, when data-copy costs are factored in. This means the communication cost on 4 processors (2*T_C) is roughly the same as the computational cost (T_1); hence a 4-processor run should be 50% efficient. The table shows the model predictions are reasonably accurate across a wide range of processor counts.

The model also indicates that the hypercube algorithm could potentially be deployed on IBM's BG/L to obtain results similar to those achieved by IBM with its 3d routing algorithm. For example, using a 4x larger value of T_1 for

```
loop 1 to M/P/Q:
    list = Q generated datums (ran values)
    loop d = 0 to log2(P)-1:
        partner = (1 << d) ^ me
        mask = set bit to select $d$th dimension of hypercube
        if partner > me:
            loop over Q datums in list:
                if datum & mask: add datum to send-list
                else: add datum to keep-list
        else:
            loop over Q datums in list:
                if datum & mask: add datum to keep-list
                else: add datum to send-list
        Send send-list to partner
        Receive list from partner and append to keep-list
        list = keep-list
    perform Q' updates on local table using datums from list
```

Figure 2. *Pseudo-code for the power-of-two version of the hypercube RandomAccess algorithm.*

Processors	1	4	16	64	200	256	300	1024	4096	8192	10240
GUPS (unoptimized)	0.0147	—	0.0105	0.0634	—	—	—	0.195	—	0.797	—
GUPS (hypercube)	0.0180	0.0377	0.0930	0.273	0.567	0.858	0.739	2.81	9.25	17.24	19.72
GUPS (model)	0.0180	0.0333	0.0867	0.257	—	0.817	—	2.71	9.26	17.26	—
Parallel Efficiency (%)	100	52.4	32.3	23.7	15.8	18.6	13.7	15.2	12.5	11.7	10.7

Table 2. *Performance of the hypercube RandomAccess algorithm on a Cray XT3 machine in giga-updates-per-second (GUPS) for varying processor counts. Parallel efficiencies are computed from the single-processor hypercube rate.*

a BG/L processor (slower computation) and a 5x larger T_C (slower communication and data copy), both of which are (roughly) consistent with other measurements of BG/L vs Red Storm performance, the model equation yields a GUPS rate of 44 on a 128K-processor BG/L machine. Thus, even though BG/L processors may be relatively slow at performing updates, using 128K processors allows for several more doublings in performance (in the large-P near-linear-scaling regime of the algorithm) than any other current machine can achieve. As noted above, one caveat to this approach on BG/L (or any other machine) would be if limited bisection bandwidth curtailed the scaling of the algorithm.

4. Additional Optimizations

The basic algorithm of the previous section can be optimized in various ways. Table 3 illustrates the effect of several optimizations we tested for runs on up to 32 processors of the Red Storm machine. All runs performed the same number of updates per processor. The listed values are CPU seconds. The timings show a decrease of roughly 20% on 32 processors (25.2 secs to 19.7 secs) which would translate to a corresponding increase in GUPS rate.

The simplest optimization (Mod1) was to pre-post receives for all $log_2(P)$ iterations before the all2all operation begins. Because Red Storm allows the application to proceed while a message is being processed, this allowed message arrivals to be overlapped with computation to help compensate for load imbalance. This gave nearly an 8% benefit for small processor counts. Mod2 achieved a modest gain by moving all updates of local variables to after the communication operations completed. Mod3 achieved another modest gain by moving the MPI_Wait for the non-blocking receives to as late as possible. This meant waiting for the local list of the next iteration to be divided into two sub-lists before determining if the data had arrived.

The next set of optimizations altered the underlying MPI library. The default MPI on Red Storm is MPICH-2, but a locally built MPICH-1.2.6 was available that has significantly lower overhead. Mod4 used MPICH-1.2.6; it provided a modest advantage. This local version of MPICH-1.2.6 has a copy buffer on the send side for very short messages so that the application buffer can be freed more

Processors	Original	Mod1	Mod2	Mod3	Mod4	Mod5	Mod6	Mod7
1	7.033	7.039	7.032	7.034	7.032	7.029	7.035	—
2	10.152	10.113	11.862	10.063	9.985	9.947	10.014	—
4	13.654	13.283	13.164	13.089	12.901	12.734	12.885	—
8	17.505	16.431	16.297	16.093	15.867	15.644	15.863	—
16	21.338	19.733	19.821	19.241	18.818	18.923	18.891	—
32	25.186	23.122	22.915	22.260	22.026	21.628	20.105	19.656

Table 3. *Performance of optimizations of the hypercube RandomAccess algorithm as compared to the original version. Values are in CPU seconds for a fixed number of total updates.*

rapidly. In a highly synchronous code such as this, that optimization is neither necessary or desirable. Turning it off in Mod5 offered another modest speed-up. With this change, it now made sense to double buffer the send-list and keep-list and use nonblocking send operations (Mod7) which provided a further 10% reduction. For completeness, Mod6 is the nonblocking send optimization without turning off the very short message optimization in the MPICH-1.2.6 library. The impact of the extra copy in the MPI library is the delta between Mod6 and Mod7.

We only implemented these additional optimizations in the simpler power-of-two version of the algorithm. On 8192 processors of Red Storm they boosted the GUPS performance from 17.23 to 19.98 GUPS, which slightly bettered the non-power-of-two result of 19.72 GUPS on 10240 processors (see Table 2). We note that further optimizations could be attempted by mapping the logical hypercube used by the algorithm to the physical topology of the interconnected processors (mesh, torus, tree, etc.), but we haven't implemented any such machine-specific optimizations.

A final issue we tested is the effect on GUPS rate of changing the benchmark specification of its look-ahead parameter Q = 1024. Note that if Q were 1, a remote update would have to be completed before a processor could generate its next index; this would be extremely challenging for an MPI-style message-passing algorithm or machine to perform well on. Conversely, if Q were huge, processors could generate and store all their indices. Then a single global communication operation could be performed (e.g. an all2all) and processors could then process their entire set of local updates. This subverts the purpose of the benchmark; 1024 was chosen by the HPCC organizers as a compromise value between these two extremes.

Table 4 lists GUPS rates for the hypercube algorithm run on 32 processors of the Red Storm machine with varying Q values. The parallel efficiency is computed from the 1-processor Opteron GUPS rate of 0.0180; for these runs a 32x larger rate would be 100% efficient. As expected, GUPS rates and parallel efficiencies rise dramatically with increasing Q and eventually asymptote when Q = 8K (message size of 32K bytes) with a corresponding parallel efficiency of 38.7%.

5. Final Thoughts

A synchronous algorithm for the RandomAccess benchmark has been presented which performs a hypercube-style all-to-all communication operation that is efficient for the message sizes and data structures specific to the benchmark. The algorithm exhibits linear scaling of its computation (GUP count) and logarithmic scaling of its communication overhead which leads to high GUPS rates on large parallel machines. We believe this algorithm would perform well when running the RandomAccess benchmark on any distributed-memory message-passing machine.

Implementations of this algorithm in C are available for download from www.cs.sandia.gov/∼sjplimp/download.html. This includes power-of-two and non-power-of-two versions, both as stand-alone single files suitable for quick testing, and also as modules that can be plugged into the official HPCC testing framework.

Finally, we note that there have been numerous efforts over the years to create efficient algorithms for collective operations including all2all [1, 2, 7]. The routines or algorithms from these packages might further boost the performance of the RandomAccess algorithms we have presented here, particularly for the non-power-of-two case, where our implementation may be sub-optimal.

6. Acknowledgments

We thank Sue Kelly for supporting this work and Bruce Hendrickson for helpful discussions. Both are staff members at Sandia. We also thank a reviewer for pointing us to the appropriate chapter of [4].

Q value	1	4	16	64	256	1024	2048	4096	8192	32768
GUPS (hypercube)	0.000627	0.00193	0.00704	0.0262	0.0803	0.158	0.189	0.210	0.223	—
Parallel Efficiency (%)	0.109	0.335	1.22	4.55	13.9	27.4	32.8	36.4	38.7	—

Table 4. *Performance of the hypercube RandomAccess algorithm on 32 processors of a Cray XT3 machine in giga-updates-per-second (GUPS) for varying "look-ahead" Q values. The allowed look-ahead has a large impact on parallel performance.*

References

[1] M. Barnett, S. Gupta, D. Payne, L. Shuler, and R. van de Geijn, "Building a High-Performance Collective Communication Library", in Proceedings of Supercomputing '94, 107–116 (1994).

[2] J. Bruck, C.-T. Ho, S. Kipnis, and D. Weathersby, "Efficient Algorithms for All-to-All Communications in Multi-Port Message-Passing Systems", in Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures, Cape Cod, NJ, 1994.

[3] J. Dongarra and P. Luszczek, "Introduction to the HPC Challenge Benchmark Suite", ICL Technical Report, ICL-UT-05-01, 2005.

[4] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving Problems on Concurrent Processors: Volume 1*, Chapter 22, Prentice Hall, Englewood Cliffs, NJ, 1988.

[5] R. Garg and Y. Sabharwal, "Analysis and Optimization of the HPCC RandomAccess Benchmark on BlueGene/L Supercomputer: Extended Version", IBM Technical Report RI-05-010, 2006.

[6] R. Garg and Y. Sabharwal, "Optimizing the HPCC Randomaccess Benchmark on BlueGene/L Supercomputer", in Proceedings of SIGMetrics/Performance '06, Saint Malo, France, 2006.

[7] A. Knies, F. Barriuso, W. Harrod, and G. Adams III", "SLICC: A Low Latency Interface for Collective Communications", in Proceedings of Supercomputing '94, 89–96 (1994).

[8] P. Luszczek, J. Dongarra, D. Koester, R. Rabenseifner, R. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi, "Introduction to the HPC Challenge Benchmark Suite", available from `http://icl.cs.utk.edu/hpcc/pubs/index.html`, March 2005.